INFORMATION AND COMPUTATION 99, 109–121 (1992)

Deterministic Simulation of a Single Tape Turing Machine by a Random Access Machine in Sub-linear Time

J. M. ROBSON

Department of Computer Science, Australian National University, Canberra, ACT 2601, Australia

The class of functions computed within any time bound greater than $n \log \log n$ by a single tape Turing machine is strictly contained within the class of functions computed by a Random Access Machine in the same time according to the logarithmic cost criterion.

1. INTRODUCTION

1.1. *Background*

Turing machines which use a single tape both for their input and as their only working space (known as 1TTMs) appear to be extremely slow for many problems compared with multi-tape Turing machines and Random Access Machines (RAMs). However, apart from a few simple problems which require quadratic time on a 1TTM but have simple linear time solutions on a multi-tape Turing machine, there have been no general results proving this appearance to be correct. Indeed when the time for a RAM is reckoned by the reasonable logarithmic cost criterion (Aho *et al.*, 1974; Cook and Reckhow, 1973), which will be adopted throughout this paper, it is not immediately obvious that all problems can be solved as fast on a RAM as on a 1TTM.

1.2. *Known Results*

If one looks at machine types apparently more powerful than the straightforward RAM, results are already known establishing that these machines can simulate a 1TTM (or even possibly a general Turing machine) in sub-linear time. Dymond (1982) shows a simulation by a parallel RAM in time $O(\sqrt{T})$ (where T is the time taken by the TM). Hopcroft *et al.* (1975) describe a method of "rectangles" which will enable a non-deterministic RAM to simulate a 1TTM in time $O(T/\log T)$. Finally,

0890-5401/92 $5.00

Robson (1984) gives a simulation by a probabilistic RAM in expected time $O(T/\log\log T)$.

For a (deterministic sequential) RAM the best known result is the Hopcroft *et al.* (1975) simulation whose time is $O(T/\log T)$ under the uniform cost criterion but $\Omega(T)$ under logarithmic cost.

1.3. *New Results*

This paper extends the probabilistic result quoted above to the deterministic case; that is, it shows a deterministic simulation of a 1TTM whose (logarithmic) cost is $O(T/\log\log T)$. In the probabilistic case it was assumed that $T \geqslant n \log n$, where n is the input length (strictly we should write $T(n)$ for the time but we abbreviate it to T); this assumption will be weakened to $T \geqslant n \log\log n$ with a more careful argument.

An immediate conclusion is that any language recognised in time T by a 1TTM is recognised in time $T/\log\log T$ by a RAM. A second, slightly less immediate, conclusion is that there are languages recognised in time T by a RAM but not by a 1TTM.

1.4. *Organisation of the Paper*

To clarify the main ideas of the simulation, we begin by presenting, in Section 2, a very simple linear time simulation using a new approach. Then Sections 3 and 4 describe the new simulation in terms of a transformation from this simple one. Section 5 analyses the time complexity of those parts of the simulation which are new and Section 6 describes briefly some features which are borrowed almost unchanged from a previous simulation (Robson, 1984).

For simplicity we assume that the machine has a binary alphabet. Since an arbitrary machine is equivalent to one with a binary alphabet which runs slower by only a linear factor, this does not affect the conclusions on the order of the simulation time.

2. A Simple Linear Time Simulation

The tape of the 1TTM is notionally divided into "blocks" whose size is b symbols, where b is an even integer approximately equal to $(\log T)/4$. The tape of the Turing machine is represented for the simulation by an array *TAPE* each of whose elements is a b-bit integer representing the contents of one block. The size of *TAPE* is $O(T/\log^2 T + n/\log T)$ because a 1TTM computation which halts after time T can only have used $O(T/\log T)$ tape outside the section containing the input.

The actions of the simulation correspond to "events" where an event is

defined as the crossing of a block boundary different from the one crossed at the previous event. Thus two events are separated by at least b steps giving $O(T/\log T)$ events. The description of the configuration is completed by three scalar variables, q, *Bno*, and d: q is the machine's state at this boundary crossing and the boundary crossed is that between blocks *Bno* and $Bno+1$ in direction d.

Between one event and the next, the Turing machine head remains in blocks *Bno* and $Bno+1$ so that the actions are determined by q, d, $TAPE[Bno]$, and $TAPE[Bno+1]$. Before the simulation proper, a four dimensional array called *LOOKUP* is established which describes the machine's actions for every quadruple (state, direction, block, block); this description is another similar quadruple giving the state and direction of the machine at the next event and the resulting configuration in the two blocks affected.

Now the simulation of the step from one event to the next is described by the following Pascal statements which are clearly easy to translate into RAM instructions (apart from the use of multi-dimensional arrays, for which see Section 6).

$move := LOOKUP[q, d, TAPE[Bno], TAPE[Bno+1]];$
$TAPE[Bno] := move.Leftblock;$
$TAPE[Bno+1] := move.Rightblock;$
$q := move.state;$
$d := move.direction;$
if $d = Left$ then $Bno := Bno - 1$ else $Bno := Bno + 1$

There are two reasons why this simulation takes time $\Omega(\log T)$ for each event and therefore $\Omega(T)$ overall. On the one hand the elements of array *TAPE* (and therefore also of *LOOKUP*) have $\Omega(\log T)$ bits and on the other hand *Bno* has $\Omega(\log T)$ bits. The next two sections describe a different way of representing the contents of the machine's tape which reduces the cost of accessing it, not at every event but at enough events to reduce the average time for these accesses by a factor of $\log\log T$. Reducing the time caused by the accesses to *Bno*, which is less interesting and less novel, is discussed briefly in Section 6.

3. Phases of the Simulation of a Block

The basic idea behind speeding up the simulation is that the current contents of a block should be represented by a number of variables, one

"large" variable of $\Omega(\log T)$ bits and some smaller ones. At most events only the smaller variables are referred to and updated but after about $\log \log T$ events in the block, the large variable also needs to be updated. The time spent in the block between two occasions when the large variable is used is a "phase" of the simulation of the block.

DEFINITION. The "superblock" of a given tape block is the section of tape of size $2b$ symbols centred on the block; that is, it consists of the block itself and the adjoining $b/2$ symbols of each neighbouring block.

DEFINITION. The "time attributed" to a tape block consists of every Turing machine step after which the read/write head is within the superblock of the block.

Thus each Turing machine step is attributed to exactly two blocks.

DEFINITION. Given the superblock surrounding a block at an event, a "feasible" block is a block which might arise in the same position at a subsequent event with at most $(b \log b)/(3 \log S)$ steps attributed to the block between the two events (where S is the number of states of the Turing machine).

LEMMA. *For a given superblock, the number of feasible blocks is* $b^{1/3 + o(1)}$.

The proof of this lemma is deferred to an appendix but a simple argument shows its plausibility: to carry $\log S$ bits of information into the block from outside the superblock, the machine needs b steps ($b/2$ on the way in and again on the way out); thus within the stated number of steps, only $(\log b)/3$ bits can have been carried in.

DEFINITION. The computation in a block is divided into "phases" where the first phase starts at the first event in the block and a phase ends at the last event in the block or at the next event which produces a configuration in the block which is not a feasible block with respect to the superblock which surrounded the block at the start of the phase.

Since a phase which ends with a non-feasible block arising must include at least $(b \log b)/(3 \log S)$ steps attributed to the block, the total number of phases over the whole computation is bounded by $2T/(b \log b/3 \log S) +$ (number of blocks on the tape) which is $O(T/\log T \log \log T)$.

4. A Representation of the Tape Allowing Fast Simulation

4.1. *The Representation*

Now we can be more precise about the large and small variables representing a block mentioned at the start of Section 3. The large variable for a block contains a $2b$ bit number giving the superblock's contents at the start of the current phase. The small variables include one which is the index of the current block configuration in a list of all feasible blocks for this superblock. By the lemma the size of this small variable is $O(\log b)$ bits. The actual value of the current block configuration could be looked up in a two dimensional array *FEASIBLE*, though this is not necessary for the simulation of most events. Thus the relation between the simple simulation of Section 2 and the new one could be expressed as

$$TAPE[i] = FEASIBLE[SUPERBLOCK[i], index[i]]$$

The representation of the tape is completed by a third variable, associated with the boundary between each pair of adjacent blocks; $BOUNDARY[i]$ contains a value dependent on $SUPERBLOCK[i]$ and $SUPERBLOCK[i+1]$ in a way to be explained in Section 4.2.

4.2. *Simulating Normal Events*

We designate an event as "normal" if it is internal to a phase of each of the two blocks on whose boundary it occurs. This term could be regarded as inaccurate since there is no guarantee that any normal events occur (if the number of steps between events is often very large). At a normal event it is necessary to take a quadruple (index, index, state, direction) indicating the two blocks and the Turing machine's behaviour and produce a new similar quadruple indicating the changed blocks and the machine's behaviour at the next event. As long as the *SUPERBLOCK*s of the two blocks on each side of this boundary remain the same, this is achieved by a particular function from quadruples to quadruples. The *BOUNDARY* value encodes this function, containing the function values for all possible arguments packed into a single integer.

The unpacking of the required quadruple from the boundary value is done by lookup in a fixed array *SELECT* and the components of the quadruple are themselves selected by lookup in four more fixed arrays, giving as the equivalent of the Pascal statements of Section 2

$move := SELECT[q, d, INDEX[Bno], INDEX[Bno+1], BOUNDARY[Bno]];$

$Leftindex := LIND[move];$

$Rightindex := RIND[move];$

if $(Leftindex = 0)$ or $(Rightindex = 0)$ then *Endofphase*

```
else begin
    INDEX[Bno] := Leftindex;
    INDEX[Bno + 1] := Rightindex;
    q := STATE[move];
    d := DIRECTION[move];
    if d = Left then Bno := Bno − 1 else Bno := Bno + 1
    end
```

where an index value of zero has been used to flag a non-feasible block.

4.3. *Updating at an End of Phase*

At the end of a phase the array *FEASIBLE* is used to find the actual current contents of the blocks concerned and then these are used to simulate the step to the next event and to update all affected *SUPERBLOCK* and *BOUNDARY* values. Two further fixed arrays are used for this:

SB[*B*1, *B*2, *B*3] gives the superblock corresponding to block *B*2 with *B*1 and *B*3 to its left and right,

BDRY[*SB*1, *SB*2] gives the boundary value for the boundary with superblocks *SB*1 and *SB*2 to its left and right.

Thus the Endofphase procedure is

```
Lblock := FEASIBLE[SUPERBLOCK[Bno], INDEX[Bno]];
Rblock := FEASIBLE[SUPERBLOCK[Bno + 1], INDEX[Bno + 1]];
move := LOOKUP[q, d, Lblock, Rblock];
{exactly as in Section 2}
if Leftindex = 0 then
  begin
  Farleft := FEASIBLE[SUPERBLOCK[Bno − 1],
             INDEX[Bno − 1]];
  SUPERBLOCK[Bno] := SB[Farleft, move.Leftblock,
                        move.Rightblock];
  BOUNDARY[Bno − 1] := BDRY[SUPERBLOCK[Bno − 1],
                            SUPERBLOCK[Bno]];
  INDEX[Bno] := 1
  end
  else INDEX[Bno] := Leftindex;
```

```
if Rightindex = 0 then
  begin
  Farright := FEASIBLE[SUPERBLOCK[Bno + 2],
                       INDEX[Bno + 2]];
  SUPERBLOCK[Bno + 1] := SB[move.Leftblock,
                            move.Rightblock, Farright];
  BOUNDARY[Bno + 1] := BDRY[SUPERBLOCK[Bno + 1],
                            SUPERBLOCK[Bno + 2]];
  INDEX[Bno + 1] := 1
  end
  else INDEX[Bno + 1] := Rightindex;
BOUNDARY[Bno] := BDRY[SUPERBLOCK[Bno],
                      SUPERBLOCK[Bno + 1]];
q := move.State;
d := move.Direction;
if d = Left then Bno := Bno - 1 else Bno := Bno + 1
```

4.4. *Setting up the Fixed Arrays*

A number of fixed arrays have been mentioned as being used in the simulation proper and must be set up in a preliminary stage. They are

$SELECT[1..S, Left..Right, 1..Maxindex, 1..Maxindex, 0..Maxboundary]$,

$LIND, RIND, STATE, DIRECTION[0..Maxquad]$,

$SB[0..Maxblock, 0..Maxblock, 0..Maxblock]$,
$BDRY[0..Maxsuperblock, 0..Maxsuperblock]$,

$FEASIBLE[0..Maxsuperblock, 1..Maxindex]$

and

$LOOKUP[1..S, Left..Right, 0..Maxblock, 0..Maxblock]$,

where *Maxindex*, *Maxboundary*, *Maxblock*, *Maxsuperblock*, and *Maxquad* are respectively the largest integers representing an index, a boundary, a block, a superblock, and a quadruple (state, direction, index, index):

$$Maxblock = 2^b - 1,$$

$$Maxsuperblock = 2^{2b} - 1,$$

$$Maxindex = b^{1/3 + o(1)},$$

$$Maxquad = 2SMaxindex^2 - 1,$$

$$Maxboundary = (Maxquad + 1)^{Maxquad + 1} - 1.$$

Of these arrays *SELECT*, *LIND*, *RIND*, *STATE*, *DIRECTION*, and *SB* are simply to provide shifting and masking operations which are not in the RAM instruction set. These are all easily set up by loops whose structure is independent of the details of the Turing machine.

Each element of *LOOKUP* is indexed by a (state, direction, block, block) quadruple and describes the outcome if the machine crosses the boundary between the two blocks in the given state and direction. The computation on this pair of blocks is simulated until one of four things happens:

(i) the computation leaves the pair of blocks: the outcome is recorded in *LOOKUP* immediately,

(ii) the computation recrosses the boundary between the two blocks: this establishes a different (state, direction, block, block) quadruple which is recorded as the "successor" of the original one,

(iii) the number of simulated steps exceeds $2^b bS$ indicating looping within the block,

(iv) the computation halts: this fact is recorded together with whether the input was accepted.

Then for any (state, direction, block, block) whose outcome is not yet known, the chain of successors is followed until either it loops or a known outcome is found and then this outcome is recorded for every member of the chain.

To find the list of *FEASIBLE* blocks for each superblock, we consider every (superblock, direction) pair and simulate S computations entering the superblock in this direction (one computation for each state) for up to $(b \log b)/(3 \log S)$ steps. Any event occurring during this simulation gives a feasible block for this pair and this block is recorded as feasible together with the time taken to reach it. If the computation leaves the superblock, we know that when it returns, the superblock contents will be unchanged and the direction will be reversed; this (superblock, direction) pair is recorded as a "successor" of the original one together with the time spent in the superblock. Then each pair has its list of known feasible blocks extended to include the known feasible blocks of its successors provided the total time in the superblock to reach them is $\leqslant (b \log b)/(3 \log S)$. This is continued until some sweep finds no new information about feasible blocks and then the two lists of feasible blocks for each superblock are merged to form the appropriate row of *FEASIBLE*.

Finally *BDRY* is set up. For every pair of compatible superblocks ("compatible" meaning that the one set up later must contain half of a feasible block of the other in its overlap with the central block of the other), $Maxquad + 1$ values are computed and packed together to form the

appropriate element of *BDRY*. Each value is computed by considering the actions of the Turing machine after crossing the boundary between the blocks of the two superblocks with some quadruple (state, direction index, index) describing the machine and the two blocks:

(a) *FEASIBLE* gives the actual contents of the two blocks,

(b) *LOOKUP* gives the actual result of the computation,

(c) the indices corresponding to the resulting blocks are found by searching in the appropriate rows of *FEASIBLE*,

(d) the resulting state, direction, and two indices are packed together to give the required value.

5. Timing

5.1. *Precomputation*

The first thing the simulating program does is read its input. As the input is read, it is packed "on the fly" so that when n bits have been read, they are packed into integers of about $(\log n)/4$ bits. Provided the packing is done by the algorithm mentioned in the next section, the time to do this is $O(n \log \log n)$. If T is easy to compute, then the simulation now proceeds with the correct value of T; otherwise a succession of values T_i are chosen (T_0 is the first power of 2 exceeding n, $T_{i+1} = 2T_i$) until the simulation assuming $T = T_i$ does halt within T_i simulated steps. The total time taken by the unsuccessful simulations will not exceed twice that taken by the successful one.

Of the fixed arrays which must be set up before the simulation proper, those which do not depend on the details of the Turing machine, namely *SELECT*, *LIND*, *RIND*, *STATE*, *DIRECTION*, and *SB*, can all be easily set up in time

$$O(\text{Number of elements} \times \text{Polynomial}(\log(\text{Maximum element or subscript})))$$
$$= O(2^{3b} \times \text{Polynomial}(b))$$
$$= o(T/\log \log T).$$

The $O(2^{2b})$ simulations in setting up *LOOKUP* each take time $O(2^b \times \text{Polynomial}(b))$ and the process of following chains of successors to complete *LOOKUP* takes time $O(2^{2b} \times \text{Polynomial}(b))$ since a link from an element of a chain to its successor is followed only twice. Thus the total time for *LOOKUP* is again $O(2^{3b} \times \text{Polynomial}(b))$.

The $O(2^{2b})$ simulations in setting up *FEASIBLE* each take time

$O(\text{Polynomial}(b))$ and the process of combining the known feasible blocks of each superblock's successors with its own known feasible blocks must terminate after at most $(b \log b)/(3 \log S)$ iterations giving a total time of $O(2^{2b} \times \text{Polynomial}(b))$.

BDRY has 2^{4b} elements but the only ones ever set up or accessed are those whose two subscripts are compatible, effectively reducing the number of elements to $O(2^{3.5b} \times \text{Polynomial}(b))$. Since the elements are bounded by *Maxboundary* and

$$\begin{aligned} \log Maxboundary &= O(Maxquad \log Maxquad) \\ &= O(b^{2/3+o(1)} \log b) \\ &= O(b^{2/3+o(1)}), \end{aligned}$$

the time to set up each element of *BDRY* is $O(\text{Polynomial}(b))$ giving a total of $O(2^{3.5b} \times \text{Polynomial}(b)) = o(T/\log\log T)$.

Hence the total precomputation time is $o(T/\log\log T)$.

5.2. *The Simulation Proper*

As noted in Section 3, the number of phases is $O(T/\log T \log\log T)$ and all the variables used in simulating a phase's start or end have $O(\log T)$ bits. Thus the total time in these steps is $O(T/\log\log T)$. Asymptotically this will dominate the total simulation time.

The number of normal events is at most T/b and all the variables used in simulating a normal event, except the block number *Bno*, have $o(b^{2/3+o(1)})$ bits, giving a total time for these simulations of $o(Tb^{o(1)-1/3}) = o(T/\log\log T)$, excluding only the time spent incrementing and decrementing *Bno* and using it as a subscript for *INDEX* and *BOUNDARY* a total of $O(T/\log T)$ times.

6. Cutting the Cost of Manipulating Large Subscripts

We now outline briefly the method described fully in (Robson, 1984) of reducing the RAM time spent in handling large arrays of relatively small elements when references to the array elements show suitable locality.

Suppose that X is an array of $O(T)$ elements, each of $O(\log^{1-\varepsilon} T)$ bits, and that references to X are all via a subscript i which is only updated by incrementing or decrementing it by 1. Then a different representation of X reduces the average cost of an operation on i or a reference to X from $O(\log T)$ to $O(\log^{1-\varepsilon} T \log\log T)$.

Elements of X are grouped into "pages" of $p = \Omega(\log^{\varepsilon} T)$ so that the total number of bits in a page is approximately $(\log T)/2$. Two of these pages are

called "current," namely those on each side of the last page boundary crossed by the value of the subscript i. The elements of X within these current pages are kept in a small array of $O(\log T)$ elements so that they are accessed fast. Each other page has all its elements packed together into a single integer and stored in a larger array indexed by page number. The variable i is replaced by three variables, namely *pagestart*, the index (in X) of the first element of X within the current pages, $rem = i - pagestart$, and $pageno = pagestart/p$.

Now the time to access elements of X within the current pages is $O(\log^{1-\varepsilon} T)$ but when a new page becomes current, elements of X have to be packed and unpacked. This packing and unpacking can be done in time $O(\log T \log\log T)$ by a divide and conquer method and occasions when it is necessary are separated by $O(\log^{\varepsilon} T)$ operations on i. Thus the average time for an operation on i or a reference to $X[i]$ is $O(\log^{1-\varepsilon} T \log\log T)$.

This approach can be used to handle the arrays *BOUNDARY* and *INDEX* and the variable *Bno* used to index them. The value of *Bno* is still available for use in other ways as $pagestart + rem$.

Details of this approach and of the packing and unpacking algorithms can be found in (Robson, 1984). That paper also describes one method of handling multi-dimensional arrays by address calculation; arrays of pointers to lower dimensional arrays would be equally satisfactory.

7. Conclusions

From all the previous discussion it is clear that the whole simulation time is $O(n \log\log n + T/\log\log T)$. Thus we have immediately

THEOREM 1. *Any language recognised in time $T(n)$ by a single tape Turing machine is recognised by a Random Access Machine in time (under the logarithmic cost criterion) $O(\max(T(n)/\log\log T(n), n \log\log n))$.*

If the Turing machine had a read only input tape and a separate work tape, then a similar simulation could be done incorporating into the "state" of the simulated machine an indication of the position of the input tape head. Since the simulation time was proportional to the logarithm of the number of states, this would give a simulation in time $O(\max(T(n) \log n/\log\log T(n), n \log\log n))$, an improvement only if T is at least exponential.

As a fairly simple consequence we have also

THEOREM 2. *If $T_1(n) = \Omega(n \log\log n)$ is computable by a (log cost) RAM in time $O(T_1(n))$ and $T_2(n) = o(T_1(n) \log\log T_1(n))$, then there is a*

language recognised by a RAM in time $O(T_1(n))$ but not recognised by any single tape Turing machine in time $O(T_2(n))$.

Proof. By (Cook and Reckhow, 1973) there exists a language recognised by a RAM in time $O(T_1(n))$ but not in time $O(\max(T_2(n)/\log\log T_1(n), n\log\log n))$. This language cannot be recognised by a 1TTM in time $O(T_2(n))$ because the simulation would give a RAM recogniser with time $O(\max(T_2(n)/\log\log T_1(n), n\log\log n)$.

The conclusion of Theorem 2 holds also for T_1 between n and $n\log\log n$, but for quite different reasons: by the result of (Trakhtenbrot, 1964) the languages recognised by a single tape Turing machine in time $O(T_2(n))$ are also so recognised in linear time and so are regular; but the languages recognised by a RAM in time $O(T_1(n))$ include all the regular languages and also $\{a^ib^i\}$.

The questions of whether there is a sublinear time simulation of 1TTM by TM and/or of TM by RAM and more weakly whether the increase in the set of languages recognised comes in the step from 1TTM to TM or from TM to RAM (or both) are all still open and full of interest.

Appendix

Lemma. Given a section of Turing machine tape consisting of $2b$ cells numbered 1 to $2b$, we define an "event" to be an occasion when the machine crosses one of the boundaries of the "central block" of the section, (namely the section from cells $b/2+1$ to $3b/2$ inclusive) unless, since the last event, the machine has neither crossed out of the section nor traversed the central block. Given the configuration of the whole section at a time t_0 at which an event occurs, a "feasible block" is a configuration which could occur in the central block at the time of a subsequent event at a time from t_0 up to the completion of $(b\log b)/(3\log S)$ computation steps within the section.

The number of feasible blocks for a given configuration at t_0 is $b^{1/3+o(1)}$.

Proof. Suppose the number of feasible blocks is f. We consider f computations, one producing each feasible block. For each i $(1\leqslant i\leqslant b/2)$, we consider the four boundaries L_i (defined as the boundary between cells i and $i+1$), X_i $(b/2+i$ and $b/2+i+1)$, Y_i $(b+i$ and $b+i+1)$ and R_i $(3b/2+i$ and $3b/2+i+1)$. The f computations contain on average at least $2\log f/\log S - O(1)$ steps which cross one of these four boundaries (otherwise two of the f computations would be identical in their odd numbered positions and since every crossing into the area between L_i and R_i occurs at an odd numbered position, these two computations would

produce identical results in this area contradicting the supposition that they produce different configurations between $b/2$ and $3b/2$).

Now summing over i from 1 to $b/2$, we conclude that the average number of steps within the section for the f computations is at least $b(\log f/\log S - O(1))$, but this average cannot exceed $(b \log b)/(3 \log S)$.

Hence $b(\log f/\log S - O(1)) \leqslant (b/\log b)/(3 \log S)$

giving $\log f \leqslant \log b/3 + O(1)$ and finally $f = b^{1/3 + o(1)}$.

RECEIVED July 7, 1989; FINAL MANUSCRIPT RECEIVED October 16, 1990

REFERENCES

AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. (1974), "The Design and Analysis of Computer Algorithms," Addison–Wesley, Reading, MA.

COOK, S. A., AND RECKHOW, R. A. (1973), Time bounded random access machines, *J. Comput. System Sci.* **7**, 354–375.

DYMOND, P. W. (1981), Speedup of multi-tape Turing machines by synchronous parallel machines, *in* "Proceedings of 15th ACM Symposium on Theory of Computing," pp. 336–343.

HOPCROFT, J. E., PAUL, W. J., AND VALIANT, L. G. (1975), On time versus space and related questions, *in* "Proceedings of 16th IEEE Conference on Switching and Automata Theory," pp. 57–64.

ROBSON, J. M. (1984), Fast probabilistic RAM simulation of single tape Turing machine computations, *Inform. Contr.* **63**, 67–87.

TRAKHTENBROT, B. A. (1964), Turing machine computations with logarithmic delay, *Algebra i Logika* **3**, 33–48. [In Russian]

Printed in Belgium